MODULE 1

Environment, primitives & null safety

C# 14 Essentials TMS logic engine

TmsCore / Program.cs

```
Console.WriteLine("=== TMS Core ===");
string student = "Kidane";
decimal fee = 1999.99m;
```

Your first C# program

Outcomes

By the end of this session, you will:

- Set up a .NET 10 project from the command line
- Understand why C# uses static typing and what that prevents
- Choose the correct numeric type for financial vs scientific data
- Write null-safe code using `?`, `??`, and `?.`
- Run your first TMS prototype

Who uses C# and why should you care?

C# in the Real World

Microsoft

Azure, Visual Studio, .NET runtime itself

Stack Overflow

Entire backend runs on ASP.NET Core

Unity

70% of mobile games are built with C#

Banks, governments, enterprises

C# dominates where correctness matters

.NET 10 Install and verify

Pre-flight check

```bash
# Verify .NET 10 installation
dotnet --version
# Expected output: 10.0.xxx

# Create the TMS project
dotnet new console -n TmsCore --framework net10.0

# Observe what was created
cd TmsCore
dir
# Program.cs  your entry point
# TmsCore.csproj  the project configuration

# Run it
dotnet run
# Output: Hello, World!
```

What just happened

1

dotnet new console

Created a ready-to-run application

2

Program.cs

Your entry point one file, no ceremony

3

dotnet run

Compiled AND executed in one step

The project file

TmsCore.csproj Your configuration

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>

    <TargetFramework>net10.0</TargetFramework>

    <LangVersion>14</LangVersion>  <!-- C# 14 is default for .NET 10 -->

    <Nullable>enable</Nullable>     <!-- Null safety is ON -->

  </PropertyGroup>
</Project>
```

Three lines that matter

Setting	What it does
`net10.0`	Targets the latest Long-Term Support release
`LangVersion 14`	Enables C# 14 features like the `field` keyword
`Nullable enable`	The compiler warns when you risk a null crash

The decimal trap

A bug that costs real money

Financial and audit contexts: the type you pick is not a style choice.

THE BUG: Using double for money

C#

```
// THE BUG: Using double for money

double grantPerStudent = 1999.99;

double totalStudents = 100_000;

double totalBudget = grantPerStudent * totalStudents;


Console.WriteLine($"Total budget (double): {totalBudget}");

// Output: 199999000.00000003
```

Where did the `.00000003` come from?

- IEEE 754 floating-point cannot represent `1999.99` exactly in binary
- The error is tiny per operation but accumulates across thousands of transactions

THE FIX: Using decimal for money

C#

```
// THE FIX: Using decimal for money

decimal grantPerStudentFixed = 1999.99m;

decimal totalStudentsFixed = 100_000m;

decimal totalBudgetFixed = grantPerStudentFixed * totalStudentsFixed;


Console.WriteLine($"Total budget (decimal): {totalBudgetFixed}");

// Output: 199999000.00  EXACT
```

Discussion precision in real systems

"Has anyone seen a bug caused by rounding or precision in a real system? What happened? How was it discovered?"

When to use each numeric type

The decision table

Type	Size	Use in TMS	When to use
`int`	32-bit	Enrollment count, student age	Whole numbers within +/- 2 billion
`long`	64-bit	Global student ID	Very large whole numbers
`double`	64-bit	GPS coordinates	Scientific/geometric data ONLY
`decimal`	128-bit	Course fees, grant allocations	Any financial calculation
`bool`	8-bit	`IsEnrolled`, `IsGraduated`	True/false flags
`string`	Variable	Names, emails, descriptions	Text data

The rule

- Money or audit trail? Use `decimal`
- GPS or physics? Use `double`
- Everything else numeric? Start with `int`

String interpolation The professional way

Formatting output

```C#
string studentName = "Abeba";

int enrollmentCount = 3;


// String interpolation  clean, readable

Console.WriteLine($"Student: {studentName}, Courses: {enrollmentCount}");


// String concatenation  avoid this

Console.WriteLine("Student: " + studentName + ", Courses: " + enrollmentCount);
```

- Why interpolation wins: reads left to right, like natural language
- No hidden `ToString()` calls to forget
- Works with format specifiers: `$"GPA: {gpa:F2}"` gives "GPA: 3.50"

The million-dollar bug fix

C# solves the billion-dollar mistake

Context

Tony Hoare, who invented null references, called them his "billion-dollar mistake." C# fixes this with nullable reference types.

C#

```
string name = "Abeba";       // Guaranteed non-null

string? region = null;        // Explicitly nullable  the '?' is the contract


// Without null check  compiler warning CS8602

// Console.WriteLine(region.ToUpper()); // WARNING: possible null reference


// With null-conditional operator

Console.WriteLine(region?.ToUpper() ?? "NO REGION ASSIGNED");
```

The operators

Operator	Name	What it does
`?` after type	Nullable marker	"This value might be null"
`?.`	Null-conditional	"Access member only if not null"
`??`	Null-coalescing	"Use this default if null"
`??=`	Null-coalescing assignment	"Assign only if currently null"

Discussion null at runtime

"What is the worst thing that happens when you access a null variable at runtime? Has anyone experienced a crash caused by null?"

Practical null safety

Example

```
// A student might not have a region assigned yet

string? region = null;

// Safe access with a default

string displayRegion = region ?? "Unassigned";

Console.WriteLine($"Region: {displayRegion}");

// Output: Region: Unassigned

// Null-conditional: call method only if not null

int? length = region?.Length;  // length is null, not a crash

// Practical pattern: Console.ReadLine() returns string?

string? input = Console.ReadLine();

if (input is null)
{
    Console.WriteLine("No input provided.");
    return;
}

// After the null check, 'input' is promoted to non-null string
Console.WriteLine($"You entered: {input.ToUpper()}");
```

Key insight flow analysis

Key insight

After a null check, C# promotes the variable. Inside the `if` block after checking `input is null`, the compiler knows `input` is non-null on the remaining branch. This is called flow analysis.

Top-level statements

No boilerplate, just code

Program.cs

```
// Program.cs  the ENTIRE file
// No class, no Main method, no namespace ceremony

Console.WriteLine("=== TMS Core Engine ===");

string studentName = "Kidane";
int courseCount = 5;

Console.WriteLine($"Welcome, {studentName}. You are enrolled in {courseCount} courses.");

// C# 12+ collection expression
string[] courses = ["C# Essentials", "TypeScript", "ASP.NET Core"];
Console.WriteLine($"First course: {courses[0]}");
```

What is happening behind the scenes?

Compiler reality

The compiler wraps this in a `class Program` with a `static async Task Main(string[] args)` for you. You get the simplicity without losing the structure.

Digit separators and modern syntax

Readability features

```C#
// Without separators  how many zeros?
int population = 100000000;

// With digit separators  clearly 100 million
int populationClear = 100_000_000;

// Same value, same type, same IL code
// The underscore is purely for human readers
```

Quick check answer

100_000 and 100000 same type? Same value? Yes. The underscore is a digit separator for readability. It has no effect on the value or type. Both are `int`. The compiler ignores underscores entirely.

Session 1 summary

What you learned

End-of-session recap: one row per main idea tie each to your M1 lab checkpoints.

Concept	Key takeaway
.NET 10 CLI	`dotnet new`, `dotnet run` your daily tools
Static typing	The compiler catches type errors before your users do
`decimal` vs `double`	Money = `decimal`. GPS = `double`. No exceptions.
Null safety	`string` vs `string?` is your contract with the compiler
`?.`, `??`, `??=`	The three operators that eliminate null crashes
Top-level statements	Simple entry point for learning; class-based comes in M4

Hiring signal

Every .NET interview starts with: "What is the difference between value types and reference types?" and "Why would you use `decimal` instead of `double`?" You just learned both.